The Implementation of an HSM-Based Smart Meter for Supporting DLMS/COSEM Security Suite 1

Tzu-Hsuan Huang[1], Chun-Tsai Chien[2], Chien-Lung Wang[2] and I-En Liao[1]
[1]*Department of Computer Science and Engineering, National Chung Hsing University, Taichung, Taiwan*
[2]*Taiwan Information Security Center at NCHU, National Chung Hsing University, Taichung, Taiwan*

Keywords: Smart Meter, Hardware Security Module, DLMS/COSEM, Security Suite 0, Security Suite 1.

Abstract: To mitigate the impacts of climate change, many governments are making efforts to increase electricity generation from renewable sources. However, the massive amount of distributed energy resources (DER) involved introduces many challenges to electricity grid management. In the last decade, we have witnessed power grids gradually evolving to become smart grids with advanced metering infrastructure (AMI). The two-way nature of communication between smart meters and energy suppliers inevitably increases cyberattack surfaces for smart grids. As a result, cybersecurity problems associated with smart meters and smart grids are of great concern.

The purpose of this paper is to develop a smart meter using a hardware security module (HSM) that supports the security mechanisms specified in Security Suite 1 of DLMS/COSEM. To the best of our knowledge, our smart meter prototype is the first published implementation using HSM. This also represents an important step in developing more secure IoT devices in general and smart meters in particular. Our implementation is based on the open-source project GuruX, available on GitHub. We revised the smart meter program GuruxDLMS.c to run on the Nuvoton M2354 hardware security module with the ability to invoke ECDSA, ECDH, and SHA-256 functions implemented on the HSM. The smart meter developed in this research is also tested for the implementations of ECDSA with P-256, ECDH with P-256, and SHA-256 using Conformance Test Tool version 3.1 (CTT v3.1).

1 INTRODUCTION

The Glasgow Climate Pact delivered at COP26 declared a near-global net zero carbon goal by 2050. Among the mitigation strategies for reducing greenhouse gas emissions, energy supply transformations, especially phasing down coal power and speeding up the transition to clean energy and electric vehicles, are the most important issues (United Nations Framework Convention on Climate Change [UNFCCC], 2021). Therefore, we can expect that smart meter rollout will be accelerated, and the integration of massive distributed energy resources (DER) into the smart grid is a must.

The traditional power grid consists of centralized power generation plants, transmission, distribution, and substations, usually operating under security protection within an isolated network. In contrast, a modern smart grid will include front-of-the-meter (FTM) and behind-the-meter (BTM) energy systems such as wind turbines, solar panels, and power storage systems (International Energy Agency [IEA], 2022).

With hundreds of thousands of smart meters on the end-user side and massive interconnected DERs, a smart grid increases available cyberattack surfaces dramatically. In this paper, we focus on enhancing the security of smart meters, since they are located on user premises and are accessible to potential hackers.

To address the interoperability, efficiency, and security issues of smart meters, the Device Language Messaging Specification User Association (a.k.a. DLMS UA) defines DLMS/COSEM (Companion Specification for Energy Metering) for managing smart meters. DLMS/COSEM has also been adopted by international standards bodies and became the standard for IEC 62056, ANSI C12, and EN 13757-1 (DLMS User Association, 2022a). In DLMS/COSEM, three security suites that define the set of security algorithms used are provided, namely Security Suite 0, Security Suite 1, and Security Suite 2 (Kozole and Kmethy, 2019).

Huang, T., Chien, C., Wang, C. and Liao, I.
The Implementation of an HSM-Based Smart Meter for Supporting DLMS/COSEM Security Suite 1.
DOI: 10.5220/0011825300003482
In *Proceedings of the 8th International Conference on Internet of Things, Big Data and Security (IoTBDS 2023)*, pages 123-130
ISBN: 978-989-758-643-9; ISSN: 2184-4976

Most of the smart meters deployed so far support the security mechanisms in Security Suite 0, including Advanced Encryption Standard-Galois/Counter Mode-128 (AES-GCM-128) for authenticated encryption and AES-128 key wrap for key transport. This is evidenced by the majority of smart meter products certified by DLMS UA being Security Suite 0 compliant. Even though authentication can be done using GMAC in Security Suite 0, the increasing demand for communicating with smart meters to ensure secure and efficient energy management requires stronger authentication mechanisms such as ECDSA (Elliptic Curve Digital Signature Algorithm), ECDH (Elliptic Curve Diffie–Hellman key exchange), and SHA (Secure Hash Algorithm); these are defined in Security Suite 1.

In this paper, we develop a smart meter using a hardware security module (HSM) that supports the security mechanisms specified in Security Suite 1. The smart meter developed in this research is based on the open-source software GuruxDLMS.c (GuruX, 2022) with several modifications to enable it to run on the Nuvoton M2354 hardware security module (HSM); it is also tested for the implementations of ECDSA with P-256, ECDH with P-256, and SHA-256 using Conformance Test Tool version 3.1 (CTT v3.1).

The rest of this paper is organized as follows. Section II discusses related work on smart meters and smart grids as well as their security issues. Section III describes the testbed and test tool for our research. Section IV provides more detail on how GuruxDLMS.c was modified for invoking ECDSA, ECDH, and SHA-256 functions implemented on the Nuvoton M2354 HSM. Section V concludes our research results.

2 RELATED WORK

A smart grid with large-scale integration of DER will increase cyberattack surfaces. Qi et al. (2016) discuss the cybersecurity issues of integrated DER and propose a holistic attack-resilient framework to protect the power grid. They also identify some important attack scenarios against DER and suggest that more research is needed to explore how trusted platform modules (TPMs) and trusted execution environments (TEEs) can be used in DER devices.

As a standard language for smart devices, DLMS/COSEM specifies a data model, an application-layer protocol, and media-specific communication profiles for smart metering and control across electricity, gas, heat energy, water, and so on (DLMS User Association, 2022b). The specifications for DLMS/COSEM are found in two colored books, the so-called Blue Book and Green Book. The COSEM object-oriented data model and the object identification system (OBIS) are specified in the Blue Book, whereas the application layer, the lower layers, and the communication profiles are specified in the Green Book. The latest versions of the two books are Edition 14 for the Blue Book and Edition 10 for the Green Book. The information security features are defined in the Green Book.

Over 1500 DLMS-certified meter types are currently used in more than 60 countries. We therefore review several research results that pinpoint the potential vulnerabilities in DLMS/COSEM specifications.

Dantas et al. (2014) in an early paper, developed an automated tool called eFuzz for security assessments of DLMS/COSEM smart meters. The security analysis is based on the specifications in the DLMS/COSEM Green Book (Edition 7), in which AES is the primary authenticated encryption algorithm. Their experiments showed that eFuzz is an effective tool for security inspections for smart meters.

Mendes et al. (2018) developed an open-source tool called ValiDLMS for validating and auditing security of DLMS/COSEM implementations using power-line communication. ValiDLMS consists of three layers: the DLMS/COSEM environment, interaction, and testing. The security analysis was performed by employing fuzzing techniques and vulnerability tests. Their experiments found security flaws in the Low-Level Security (LLS) implementation of the smart meter provided by their industrial partner.

Luring et al. (2018) performed by-hand analyses on security aspects of the Green Book (Edition 8). They identified several vulnerabilities and suggested some effective countermeasures. In the COSEM data model, a smart meter acts as the server, and any application acting as the client that needs to access the smart meter should first establish an application association (AA). Authentication is therefore very important in the AA process. DLMS/COSEM defines three security levels for authentication, namely No Security, Low-Level Security (LLS), and High Level Security (HLS). In HLS, five methods are provided: MD1, SHA1, GMAC, SHA2, and ECDSA. The authors in (Luring et al., 2018) suggested that ECDSA provides the most secure authentication out of these methods.

The widespread use of IoT devices raises great concerns about cyber threats to resource-constrained

devices. This results in more and more IoT devices being implemented with hardware security modules (HSMs). Sklavos et al. (2017) gave a comprehensive introduction to hardware security in their edited book. The authors describe the attacks such as fault attacks and side-channel attacks and give some countermeasures against those attacks. Physically Unclonable Functions (PUFs) are also presented for improving hardware security in the IC level.

Luo et al. (2022) proposed a security framework for IoT devices using TrustZone-M-enabled MCUs and presented security analysis for potential runtime software security issues such as stack-based buffer overflow (BOF) attack, return-oriented programming attack, heap-based BOF attack, format string attack, and attacks against non-secure callable functions. Their experimental results showed that even though the HSM-like ARM Cortex-M23 provides another layer of security protection in hardware, any IoT device developed using HSM without careful programming could still suffer attacks against vulnerabilities.

3 THE TESTBED AND TEST TOOL

The testbed for our implementation consists of Nuvoton NuMicro M2354 and Conformance Test Tool (CTT) V. 3.1, as shown in Figure 1. It is a client-server model in which the CTT serves as the client and the smart meter is the server. A modified version of the GuruX smart meter program is written in M2354 to simulate a smart meter that uses a cryptographic accelerator in M2354. In this section, we describe the relevant features in Nuvoton NuMicro M2354 and then discuss how the CTT was used to test the smart meter.

3.1 Nuvoton NuMicro M2354

The NuMicro M2354 microcontroller is based on Arm Cortex-M23. In addition to the built-in TrustZone technology of the Arm v8-M architecture, it also adds protection functions against side-channel attacks and provides microcontroller platform security hardware features that allow the application system to easily realize data storage security, software execution security, and message communication security (Nuvoton NuMicro M2354 , 2022). The cryptographic accelerator in M2354 includes a secure pseudo-random number generator and supports AES, SHA, RSA, and ECC algorithms.

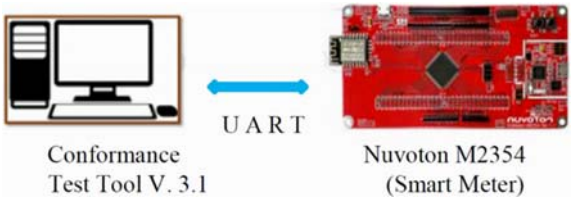

Figure 1: The testbed.

3.2 The Conformance Test Tool

The Conformance Test Tool (CTT) released by DLMS UA is a software package that implements the Abstract Test Suites (ATSs) as Executable Test Suites (ETSs). CTT acts as a DLMS/COSEM test client whereas Implementation Under Test (IUT) acts as a DLMS/COSEM server. CTT allows the selection, parametrization, and execution of the ETSs using the information taken from the Conformance Test Information (CTI) file and the information obtained by CTT from IUT (DLMS User Association, 2022). The CTT test report can be used to obtain the DLMS UA Certification.

3.3 The Test Process and Conformance Test Information

The conformance test process has three phases, namely preparation, test operations, and test report. The preparation phase involves preparation of IUT, the production of the CTI, and the preparation of the CTT. The test operations include review of the CTI, test selection and parameterization, and one or more “test campaigns” (DLMS User Association, 2018a). A conformance test report and a log file will be produced at the end of each test campaign.

The production of the CTI is an important step before testing. The CTI includes an Implementation Conformance Statement (ICS) that identifies capabilities and options as implemented in IUT. It also has information relating to IUT and its testing environment, including addresses, timeouts, baud rates, passwords, and so on.

Here we only show parts of the CTI settings, in Figures 2 and 3. Figure 2 specifies the test options and communication profile. As in Figure 1, CTT and M2354 are connected using UART, while the High-Level Data Link Control (HDLC) is used as the communication protocol. Figure 3 shows some security settings on the smart meter application layer.

```
TestOptions = {
  // Choose the profile used for testing
  CommunicationProfile = HDLC
  ReferencingMethod = LONG_NAMES
}
DoNotTestCOSEM = TRUE
// override to FALSE in one of the AAs
// HDLCProfile, if needed
HDLCProfile = {
  PhysicalLayer = {
    HDLCBaud = 9600
    OpeningMode = Direct_HDLC
  }
  DataLinkLayer = {
    AddressingSchemes = [one_BYTE_ADDRESSING]
    //ServerLowerMacAddress = 0x11
    InformationFieldLength = 128
    InactivityTimeout = 20000
    InterFrameTimeout = 10000
    ResponseTimeout = 20000
    DISCToNDMTimeout = 2000
  }
}
```

Figure 2: Test options and communication profile.

In Figure 3, we specify HLS_ECDSA as the authentication mechanism and use the security algorithms in Security Suite 1. We also specify the Global Unicast Encryption Key (GUEK), Global Authentication Key (GAK), Key Encrypting Key (KEK), and some other key pairs not shown in the figure.

4 IMPLEMENTATION DETAILS USING HSM

The objective of this research is not to build a full-fledged smart meter using a hardware security module. Instead, we demonstrate the feasibility of using a cryptographic accelerator within HSM to provide more robust security features for smart meters. In this section, we detail the modifications made to the open-source software GuruxDLMS.c such that the algorithms ECDSA with P-256, ECDH with P-256, and SHA-256 provided by the HSM can be invoked for smart meters.

```
Association [3] = {
  Enabled = TRUE
  ClientSAP = 0x12
  ApplicationContextName = LONG_NAMES_WITH_CIPHERING
  //ClientUser = 1
  AuthenticationMechanismName = HIGH_LEVEL_SECURITY_ECDSA
  DoNotTestCOSEM = FALSE // test COSEM in this AA
  Secret = "\Gurux"
  SecuritySuite = 1
  ConformanceBlock = [
    SELECTIVE_ACCESS,
    GET,
    SET,
    BLOCK_TRANSFER_WITH_GET_OR_READ,
    ACTION,
    GENERAL_PROTECTION,
    BLOCK_TRANSFER_WITH_SET_OR_WRITE,
    BLOCK_TRANSFER_WITH_ACTION,
    MULTIPLE_REFERENCES]

  ClientSystemTitle = "\CTT00000"

  GUEK = "000102030405060708090A0B0C0D0E0F"
  GAK = "D0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF"
  KEK = "31313131313131313131313131313131"
```

Figure 3: Some security settings on the application layer.

In our implementation, we use Mbed Studio 1.2.1 and Keil uVision 5.31.0.0 (Nuvoton NuMicro M2354, 2022) as the development tools and modify GuruxDLMS.c with Version 20200911.1 (GuruX, 2022) for smart meters.

4.1 Modifications to Application Association

Application association (AA) refers to a logical connection between a client and a server. Establishing an AA is the first step in accessing smart meter services. AA is modelled by COSEM "Association ShortName(SN)/LogicalName(LN)" objects that hold the Service Access Points (SAPs) identifying the associated partners, the name of the application context, the name of the authentication mechanism, and the xDLMS context. Authentication takes place during AA establishment. Once the AA is established, COSEM object attributes and methods can be accessed using xDLMS services subject to the security context and access rights specified in the given AA (DLMS User Association, 2014).

During AA establishment, the CTT will send an Association Request (AARQ) to the server, which will reply with an Association Response (AARE). Because the cip_crypt() function in src/ciphering.c did not set the security control byte (SC) correctly, the CTT cannot receive the AARE. The Security_Suite_Id in SC should be changed to 1. With this modification, an AA can be established correctly.

4.2 Modifications to SHA-256 in GuruxDLMS.c

SHA-256 can be used as an HLS authentication method in DLMS/COSEM; it is also used as hash algorithm in ECDSA. But the implementation of SHA-256 in GuruxDLMS.c is incomplete. We first complete a correct version of SHA-256 for HLS authentication. Two files, include/mbedsha256.h and src/mbedsha256.c, are added to provide the crypto function mbedsha256_encrypt(), which in turn uses the API provided by M2354 to call the hardware accelerator SHA256. Figures 4 and 5 show the programs mbedsha256.h and mbedsha256.c, respectively. In line 49 of Figure 5, the SHA256() function of the hardware accelerator is enabled and then invoked in line 56.

include/mbedsha256.h 0 – 100644

```
+ #include "gxignore.h"
+ #ifndef DLMS_IGNORE_HIGH_SHA256
+ #ifdef __cplusplus
+ extern "C" {
+ #endif
+
+
+ #include "gxbytebuffer.h"
+
+     int mbedsha256_encrypt(gxByteBuffer* inputData, gxByteBuffer* hashed);
+
+ #ifdef __cplusplus
+ }
+ #endif
+ #endif //DLMS_IGNORE_HIGH_SHA256
```

Figure 4: Declaring the crypto function mbedsha256_encrypt().

```
int mbedsha256_encrypt(gxByteBuffer* inputData, gxByteBuffer* hashed) {
    int ret = 1;

    SHA256_Enable();

    uint32_t hash[8] = {0};

    /**
     * generate hash
     */
    SHA256((uint32_t *)((uint32_t)inputData->data), inputData->size, hash);

    /**
     * convert result to byte array
     */
    unsigned char hash_byte[32];
    if ( ( ret = uint32_to_char_array( hash_byte, sizeof( hash_byte ), hash, sizeof( hash ) ) ) != 0 )
    {
        return ret;
    }
    unsigned char digest[32];
    bb_set(hashed, digest, 32);

    SHA256_Disable();
    return ret;
}
```

Figure 5: Defining the crypto function mbedsha256_encrypt().

```
RESPONSE:
  <ActionResponse>
    <ActionResponseNormal>
      <InvokeIdAndPriority Value="C1" />
      <Result Value="Success" />
      <ReturnParameters>
        <Data>
          <OctetString Value="FF0668FD698F6A192115547523FBF11F0B3A22DE74C15C0312DEBBFF72CB2E06" />
        </Data>
      </ReturnParameters>
    </ActionResponseNormal>
  </ActionResponse>
```

Figure 6: A successful HLS SHA-256 authentication.

To complete the AA using SHA-256, a client needs to compute a hash value H1 on data M1, sending these to the server for verification. Upon receiving the messages M2 (rather than M1) and H1, the server will compute a new hash value H2 on M2. If H1 = H2, then M1 = M2, meaning that the client is authenticated. Finally, the server will do the same for the client to authenticate the server. When AA is successfully established using SHA-256, CTI will report "success" for the action response of the server, as shown in Figure 6.

4.3 Modifictions to ECDH in GuruxDLMS.c

ECDH is used for key agreement between client and server, and the agreed key can be used in ECDSA for authentication or in digital signature of COSEM data. There are two crypto functions in ECDH: getSharedSecret(), which computes the shared secret Z, and the key derivation function generateKeyKDF(), which generates the shared key. These two functions can be implemented by invoking ECC_GenerateSecretZ() in M2354, as in Figure 7.

```
int mbedecdh(gxByteBuffer* sharedSecret)
{
    int ret;

    ECC_Enable();

    /**
     * Input
     * Private key = "CE69EFE1E68415AD5B9F4C8B2F3025CB1332D0BD881073309A53A526FD3D7D
     * Public key (x) = "BC316842562AB04CA987F39FBC5368899A0F2D6059E1247B68D3DC4F26C
     * Public key (y) = "F808C68B115B3B43E7F3A23D3E5F4BB3628183615A5604E1E603C9563BB
     */
    // Private key
    char d[64];
    unsigned char d_byte[] = { 0xCE, 0x69, 0xEF, 0xE1, 0xE6, 0x84, 0x15, 0xAD, 0x5B,
    // Public key (x)
    char Qx2[64];
    unsigned char Qx2_byte[] = { 0xBC, 0x31, 0x68, 0x42, 0x56, 0x2A, 0xB0, 0x4C, 0xA
    // Public key (y)
    char Qy2[64];
    unsigned char Qy2_byte[] = { 0xF8, 0x08, 0xC6, 0xBB, 0x11, 0x5B, 0x3B, 0x43, 0xE
    // buffer used to keep Z
    char z[32];

    /**
     * convert byte array to HEX string
     */
    if ( ( ret = byte_to_hexstr( d_byte, 32, d, 64 ) != 0 ) ||
         ( ret = byte_to_hexstr( Qx2_byte, 32, Qx2, 64 ) != 0 ) ||
         ( ret = byte_to_hexstr( Qy2_byte, 32, Qy2, 64 ) != 0 ) )
    {
        return ret;
    }
    /**
     * Generate Shared Secret
     */
    if( ( ret = ECC_GenerateSecretZ( CRPT, CURVE_P_256, d, Qx2, Qy2, z ) ) != 0
    {
        return ret;
    }

    /**
     * convert HEX string to byte array
     */
    int sharedSecret_len;
    unsigned char* sharedSecret_byte = hexstr_to_byte( z, &sharedSecret_len );
    bb_set(sharedSecret, sharedSecret_byte, sharedSecret_len);

    ECC_Disable();

    return ret;
}
```

Figure 7: Implementing ECDH using M2354.

4.4 Adding ECDSA as Authentication Method to GuruxDLMS.c

As mentioned in Section II, HLS_ECDSA is the most secure authentication method in DLMS/COSEM. Table 1 shows the four passes of HLS_ECDSA authentication (DLMS User Association, 2014). In this subsection, we describe how ECDSA is implemented using M2354 during AA.

Table 1: The four passes of HLS_ECDSA Authentication (DLMS User Association, 2014).

Authentication mechanism	Pass 1: C →S	Pass 2: S →C	Pass 3: C →S f(StoC)	Pass 4: S→C f(CtoS)
	Carried by			
	AARQ	AARE	XX.request reply_to_HLS authentication	XX.response reply_to_HLS authentication
mechanism_id(7) HLS ECDSA	*CtoS*: Random string 32 to 64 octets Optionally: System-Title-C in calling-AP-title, Cert-Sign-Client in calling-AE-qualifier	*StoC*: Random string 32 to 64 octets Optionally: System-Title-S in responding-AP-title, Cert-Sign-Server responding-AE-qualifier	ECDSA(SystemTitle-C || SystemTitle-S || StoC || CtoS)	ECDSA(SystemTitle-S || SystemTitle-C || CtoS || StoC)

Legend:
- C: Client, S: Server, CtoS: Challenge client to server, StoC: Challenge server to client
- xx.request / .response: xDLMS service primitives used to access the reply_to_HLS authentication method of the "Association SN / LN" object.

src/gxinvoke.c

```
251 +       else if (settings->base.authentication == DLMS_AUTHENTICATION_HIGH_ECDSA)
252 +       {
253 +           BYTE_BUFFER_INIT(&tmp);
254 +           bb_capacity(&tmp, sizeof(settings->base.sourceSystemTitle) + sizeof(settings->base.sourceSystemT
    settings->base.stoCChallenge.size + settings->base.ctoSChallenge.size);
255 +           bb_set(&tmp, settings->base.sourceSystemTitle, sizeof(settings->base.sourceSystemTitle));
256 +           bb_set(&tmp, settings->base.cipher.systemTitle.data, settings->base.cipher.systemTitle.size);
257 +           bb_set(&tmp, settings->base.stoCChallenge.data, settings->base.stoCChallenge.size);
258 +           bb_set(&tmp, settings->base.ctoSChallenge.data, settings->base.ctoSChallenge.size);
259 +           readSecret = &tmp;
260 +
261 +           // verify signature
262 +           // e->parameters.byteArr->data
263 +           if (mbedecdsa_verify(readSecret, e->parameters.byteArr) == 0)
264 +           {
265 +               accept = 1;
266 +           }
```

Figure 8: The server verifies the signature of the client.

In Pass 3, the client needs to generate its authentication message and sign the message with its private key using ECDSA. In Pass 4, the server first verifies the received message from the client and then generate its authentication message using ECDSA. The processes of Pass 3 and Pass 4 are very similar, so we only show the code for Pass 4.

Upon receiving the client's authentication message, the server extracts the client's signature as in lines 254–259 of Figure 8 and invokes mbedecdsa_verify() in line 263 to verify the client. After the client is verified, the server goes on to generate its authentication message.

The server signs its signature by invoking mbedecdsa_sign() as in Figure 9 and sends the response to the client, which receives the correct action response as in Figure 10.

The functions mbedecdsa_sign() and mbedecdsa_verify() use the APIs provided by M2354 to call the hardware accelerators ECC_GenerateSignature() and ECC_VerifySignature() in M2354, respectively.

src/dlms.c

```
5707 +    else if (settings->authentication == DLMS_AUTHENTICATION_HIG
5708 +    {
5709 +        mbedecdsa_sign(secret, reply);
5710 +    }
```

Figure 9: The server creates its signature.

```
RESPONSE:
 <ActionResponse>
   <ActionResponseNormal>
     <InvokeIdAndPriority Value="C1" />
     <Result Value="Success" />
     <ReturnParameters>
       <Data>
         <OctetString Value="7EC6E3DD90BBD99D49A0F8D6A46D92B37D0F12A236927B2D
       </Data>
     </ReturnParameters>
   </ActionResponseNormal>
 </ActionResponse>
```

Figure 10: The client verifies the server and gets the correct action response.

Figure 11 shows the function mbedecdsa_sign(). In line 9, the hardware accelerator ECC is enabled. The function mbedsha256_encrypt() defined in Figure 5 is called in line 15 to compute the hash value, and the function ECC_GenerateSignature() with CURVE_P_256 specified as an argument is invoked in line 41.

4.5 Adding General_ciphering() and General_signing() to Protect xDLMS APDU

An application protocol data unit (APDU) is a data unit used by the application service of the DLMS protocol with extensions (xDLMS). DLMS provides two layers of protection to the APDU, signing and ciphering. To apply encryption, authentication, or authenticated encryption, the general-ciphering APDU is used. To apply digital signature, the general-signing APDU is used (DLMS User Association, 2014). In our implementation, AES-GCM-128 is the encryption algorithm for the general-ciphering APDU. The client and the server need to agree on a shared key for AES-GCM to encrypt and decrypt the APDU. This is achieved by using the ECDH algorithm in the general_ciphering() function. For the general-signing APDU, the ECDSA algorithm is used.

Suppose the client wants to read the value of one COSEM object's attributes. It will perform the

following steps to create a GET-REQUEST service APDU:

```
int mbedecdsa_sign(gxByteBuffer* auth_data, gxByteBuffer* reply_signature)
{
    int ret = 1;

    ECC_Enable();

    /**
     * Input
     */
    gxByteBuffer hash;
    mbedsha256_encrypt(auth_data, &hash);
    char g_SHA_msg[64];

    // Privat key = "CE69EFE1E68415AD5B9F4C8B2F3025CB1332DDBD881073309A53A
    char gD[64];
    unsigned char gD_byte[] = { 0xCE, 0x69, 0xEF, 0xE1, 0xE6, 0x84, 0x15,

    // random integer k
    char gK[] = "10149664D1C9595133482781 24EBF054";

    // buffer used to keep digital signature (R,S) pair
    char gR1[128], gS1[128];

    /**
     * convert byte array to HEX string
     */
    if ( ( ret = byte_to_hexstr( hash.data, 32, g_SHA_msg, 64 ) != 0 ) ||
         ( ret = byte_to_hexstr( gD_byte, 32, gD, 64 ) ) != 0 )
    {
        printf("  ! byte_to_hexstr error");
        return ret;
    }
    /**
     * Calculate ECC signature
     */
    if( ( ret = ECC_GenerateSignature( CRPT, CURVE_P_256, g_SHA_msg, gD, g
    {
        printf("ECC signature generation failed!!\n");
        return ret;
    }

    /**
     * convert HEX string to byte array
     */
    int R_len, S_len;
    unsigned char* R_byte = hexstr_to_byte( gR1, &R_len );
    unsigned char* S_byte = hexstr_to_byte( gS1, &S_len );

    bb_set(reply_signature, R_byte, 64);
    bb_set(reply_signature, S_byte, 64);

    ECC_Disable();

    return ret;
}
```

Figure 11: The function mbedecdsa_sign(), using hardware accelerator.

1. use ECDSA to compute the signature of the client's request and generate a general-signing APDU;
2. use ECDH to get the agreed key;
3. use the agreed key to encrypt the general-signing APDU;
4. send the general-ciphering APDU to the server.

Upon receiving the general-ciphering APDU of the client, the server will perform the following steps to generate a GET-RESPONSE APDU:

1. use ECDH to get the agreed key;
2. use the agreed key to decrypt the general-ciphering APDU and get the general-ciphering APDU;
3. verify the signature of the client;
4. use ECDSA to compute the signature of the server's response and generate a general-signing APDU;
5. use ECDH to get the agreed key;
6. use the agreed key to encrypt the general-signing APDU and produce the general-ciphering APDU;
7. send the general-ciphering APDU to the client.

After receiving the general-ciphering APDU from the server, the CTT successfully removes general ciphering and general signing and verifies the signature of the server.

4.6 Testing the Test Case APPL_OPEN_1

The purpose of the test case APPL_OPEN_1 is to verify that the implementation under test (IUT) is able to establish an AA with the application context, authentication mechanism, and xDLMS context declared (DLMS User Association, 2018b). APPL_OPEN_1 contains three subtests. Subtest 1 is to establish an AA using the parameters declared, returning PASSED if the AA is established. Subtest 2 is to check if the AA is in the associated state, in which CTT issues a get-request command to read the attributes of the Association LN object and tests the protection of the APDU using the encryption and authentication mechanisms declared. Subtest 3 is to release the AA, returning FAILED if the procedure fails. The test report shows that all three subtests with HLS_ECDSA and Security Suite 1 declared are passed.

5 CONCLUSIONS

Within the smart grid, smart meters can be thought of as critical IoT devices in critical infrastructure. In a world oriented toward net-zero energy transformation, smart meters may be connected to more and more services, resulting in increased cyberattack surfaces.

In this research, we employ the hardware accelerators of SHA-256, ECDH with P-256, and ECDSA with P-256 in Nuvoton M2354 to provide more secure authentication for smart meters. The implementations are also tested using the conformance test tool from DLMS UA. We anticipate that in future development, smart meters will adopt hardware security modules for more secure and robust services.

It should be noted that ECDSA is a type of Public Key Infrastructure (PKI) algorithm. The key pair of each party should come from a trusted Certificate Authority (CA). In our experiments, the key pairs of client and server are embedded in the codes for easy testing.

ACKNOWLEDGEMENTS

This research was supported in part by the Taiwan Information Security Center at NCHU (TWISC@NCHU) and the Ministry of Science and Technology, Taiwan, under grant numbers: MOST 109-2218-E-005-005 and MOST 111-2218-E-005-006-MBK. The authors are very grateful to the Taiwan Testing and Certification Center for cooperation on testing smart meters using the conformance test tool. We also like to express our deep appreciation to Nuvoton Technology Corporation for their technical support on NuMicro M2354. Without the open-source project GuruX, developing a smart meter program would be a huge challenge. We owe many thanks to the many contributors to the GuruX project.

REFERENCES

Dantas, H., Erkin, Z., Doerr, C., Hallie, R., & van der Bij, G. (2014). eFuzz: A fuzzer for DLMS/COSEM electricity meters. *In Proceedings of 2nd Workshop on Smart Energy Grid Security*, 31-38.

DLMS User Association (2014). *DLMS/COSEM architecture and protocols* [Technical Report] (Ed. 8).

DLMS User Association (2018a). *CTT 3.1 task force, DLMS/COSEM conformance testing – Abstract test plans for CTT 3.1*[Technical Report] (V. 2.2).

DLMS User Association (2018b). *DLMS/COSEM conformance testing process* [Technical Report] (Ed. 6.1).

DLMS User Association (2022a). *DLMS test platform.* https://ctt.dlms.com/home

DLMS User Association (2022b). *Overview of DLMS/COSEM*. https://www.dlms.com/dlms-cosem/overview

GuruX. (2022). GuruxDLMS.c. (Version 20200911.1) https://github.com/gurux

International Energy Agency (IEA) (2022). *Unlocking the Potential of Distributed Energy Resources*. IEA, Paris.

Kozole, M. and Kmethy, I. (2019). *Security in DLMS: A white paper by the DLMS User Association.* DLMS UA.

Luo, L., Zhang, Y., White, C., Keating, B., Pearson, B., Shao, X., Ling, Z., Yu, H., Zou, C., & Fu, X. (2022). On Security of TrustZone-M Based IoT Systems. *IEEE Internet of Things Journal,* 9(12), 9683-9699.

Luring, N., Szameitat, D., Hoffmann, S., & Bumiller, G. (2018). Analysis of security features in DLMS/COSEM: Vulnerabilities and countermeasures. *In Proceedings of IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT).*

Mendes, H., Medeiros, I., & Neves, N. (2018). Validating and securing DLMS/COSEM implementations with the ValiDLMS framework. *In Proceedings of 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W),* 179-184.

Nuvoton NuMicro M2354 (2022). https://www.nuvoton.com/products/microcontrollers/arm-cortex-m23-mcus/m2354-series/

Sklavos, N., Chaves, R., Di Natale, G., and Francesco Regazzoni. F. (2017). Hardware Security and Trust: Design and Deployment of Integrated Circuits in a Threatened Environment. Springer.

Qi, J., Hahn, A., Lu, X., Wang, J., & Liu, C. (2016). Cybersecurity for distributed energy resources and smart inverters. *IET Cyber-Physical Systems: Theory & Applications*, 1(1), 28–39.

United Nations Framework Convention on Climate Change (UNFCCC) (2021). *The Glasgow Climate Pact.* 26th UNFCCC Conference of the Parties (COP26), Glasgow, United Kingdom. https://unfccc.int/documents/310475.